John von Neumann Institute for Computing

Concurrent Parallel Shortest Path Computation

D. Nussbaum, J.-R. Sack, H. Ye

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 277-284, 2006.

http://www.fz-juelich.de/nic-series/volume33

Concurrent parallel shortest path computation *

Doron Nussbaum, Jörg-Rüdiger Sack and Hua Ye [a]

[a]School of Computer Science, Carleton University, Ottawa, Ontario K1S-5B6, Canada
e-mail: {nussbaum, sack, hye}@scs.carleton.ca

In this paper we introduce the notion of concurrent parallelism for the fundamental problem of answering shortest paths queries between pairs of points located on a weighted polyhedron or terrain. We discuss design and implementation of our algorithm and show experimentally, that the efficiency and speed-up for parallel shortest paths queries can be increased through concurrency. Our work also represents the first implementation of the bisector algorithm which currently has the best theoretical time complexity for computing weighted shortest paths.

1. Introduction

The computation of shortest paths arises in application areas such as geographical information system (GIS), network routing, and robotics. It ranks among the fundamental problems studied in computer science, graph theory and computational geometry. Our motivation came from search & rescue, where a rescue team needs to reach the location of an accident as fast as possible, i.e., using the shortest path. Typically in such problems, the domain is a terrain represented as a triangular irregular network, called TIN. The cost of travel in terrains may differ from triangle to triangle. For example, one triangle may represent water while another may represent a forest. To capture travel through such terrains, the Euclidean distance is not an appropriate distance measure. The weighted shortest path problem arises when we associate a positive weight with each triangle. Unlike in the Euclidean setting, weighted shortest path computations on terrains are both time consuming and complex to solve. No exact algorithm exists and even the first approximation algorithm [13] discovered was of limited practical value (even if it had been implemented) as the time complexity was approximately $O(n^8)$ for a TIN consisting of n weighted faces. In order to speed up the computation we follow a two-prong approach (1) developing more efficient approximation techniques (the reader is referred to the most recent work, e.g., [5], and for a comparison of existing sequential algorithms) and (2) designing parallel algorithms.

For Euclidean shortest paths computations, a variety of parallel algorithms can be found in the literature [1,6,8–10,12,16]. For weighted shortest paths computations, given the high sequential time complexities and its practical relevance, parallelization is appropriate or even required which motivates the work described here and our earlier work discussed in [12]. The differences between existing parallel algorithms are: (a) types of parallel architectures, (b) particular problem variants (e.g., one-to-one and one-to-all), (c) objects/domains, and (d) implementation. Parallel one-to-one shortest paths algorithms are usually based on a variation of Dijkstra's shortest path algorithm [7]. Such parallel algorithms, which are designed for distributed memory computers, partition the data among the processors and then execute Dijkstra's algorithm locally on each processors. When the shortest path processing reaches the boundary of a partition, processors send messages to neighbouring processors to update their local information (see [12] for more details of a parallel implementation).

The shortest path problem appears to be sequential in nature and the performance of a parallel implementation degrades as the number of processors increases [9,12]. In an information system accessed by many users, multiple weighted shortest path queries must be processed (e.g., for planning several approaches in search&rescue or in a tourist information system). Using a parallel computer these queries would be processed sequentially and the response time of each query depends on the order in which the queries arrive. Often when a shortest path query is processed on a medium to large number of processors (16-128 processors) the idle time of each processor increases. This leads to a reduction in utilization and efficiency of the processors as was observed by [9,12], where the efficiency for p processors was about $\sqrt{p}$.

This paper discusses an implementation and experimental study of concurrent parallelism for shortest path queries on weighted polyhedra or TINs. Our objective is to demonstrate an increase in processor utilization and speed-up. The paper is organized as follows: in Section 2 we sketch the bisector ϵ-approximation technique of [4,5] used in this implementation. In Section 3 we discuss concurrent parallelism. In Section 4 we show experimental results and we conclude with Section 5.

*Research supported in part by NSERC and Sun Microsystems of Canada

2. Formal Problem Definition and Background

We first define the problem formally. Let $\mathcal{P} = \{f_1, f_2, \ldots, f_n\}$ be a continuous 2.5D polyhedral surface (*terrain*) composed of n triangular faces $f_i, 1 \leq i \leq n$. Each face $f_i, 1 \leq i \leq n$, has an associated positive weight w_i which determines the cost of travel through that particular face. Given such a terrain $\mathcal{P}$, and two points s, t on $\mathcal{P}$, a weighted shortest path $\Pi(s,t)$ between s and t is defined to be a path of minimum cost among all possible paths joining s and t that lie on the surface of $\mathcal{P}$. The cost of the path $c(\Pi(s,t))$ is the sum of the Euclidean lengths of all path segments multiplied by the corresponding face weight $c(\Pi(s,t)) = \sum_{i=1}^{k}(|s_i| \times w(f(s_i)))$, where k is the number of segments of the path; $|s_i|$ is the Euclidean length of the i^{th} segment s_i; $f(s_i)$ is the face that contains segment s_i and $w(f(s_i)))$ is the weight of face $f(s_i)$. (When all weights are equal the Euclidean shortest path problem is obtained.)

The determination of weighted shortest paths on polyhedral surfaces (or in 3-space) has been studied extensively by researchers over the past approximately 20 years; only approximation solutions are available and some problem instances are known to be $\mathcal{NP}$-complete. Most of the currently best approximation algorithms utilize a discretization approach. They convert the continuous problem into a discrete problem creating an approximation graph $G^* = (V^*, E^*)$; and then execute Dijkstra's shortest path algorithm [7] between two points in G^* corresponding to the two inputs points on the surface. The approach must guarantee that the *approximating path* so constructed provides an approximation to the true shortest path. The approximation path is called ϵ-*approximation*, $\epsilon \geq 0$, of the true shortest path, if its cost is at most $(1+\epsilon)$ times the cost of the real shortest path.

We briefly sketch the discretization of $\mathcal{P}$: first Steiner points are strategically added to $\mathcal{P}$ which, together with the vertices of $\mathcal{P}$, form the vertex set V^*. Once V^* has been created, the edge set E^* is defined by interconnecting vertices of V^*. The choice of interconnection pattern and its size are crucial. Typically edges connect (selected) pairs of vertices located within one face or on at most two adjacent faces. The cost associated with each such edge then corresponds to the cost of the weighted shortest path between the two points restricted to lie inside a face (or inside two adjacent faces).

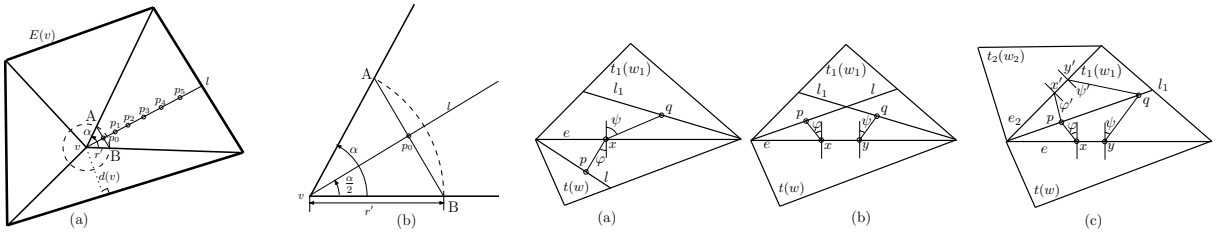
The performance of an approximation algorithm in terms of execution time and path cost accuracy is strongly correlated to the number of Steiner points in V^* and the edge interconnect scheme. In general, when the number of Steiner points in G^* increases, the accuracy of the approximated solution improves but the execution time increases. The challenging task is to develop very efficient discretization techniques while maintaining a high accuracy of approximation.

For weighted shortest paths on a non-convex polyhedral surface, Mitchell et al. [13] first presented an $O(n^8 \log \frac{n}{\epsilon})$ time ϵ-approximation algorithm using what they call "continuous Dijkstra" paradigm. Lanthier et al. [11] and Aleksandrov et al. [2,3] adopted a natural approach to discretize the surface by placing Steiner points on the TIN edges (see Section 2.1 for details). Lanthier et al. [11] produce an approximate shortest path $\Pi'(s,t)$, which holds $||\Pi'(s,t)|| \leq \beta(||\Pi(s,t)|| + W|L|),\ \beta > 1$ where W is the maximum weight among all face weights of the surface and $|L|$ is the Euclidean length of the longest edge of the surface. It runs in $O(n^3 \log n)$ time in the worst case, but performs very well in practice (see [11] and [17]). The ϵ-approximation algorithm presented by Aleksandrov et al. [3] runs in $O(\frac{n}{\epsilon} \log \frac{1}{\epsilon}(\frac{1}{\sqrt{\epsilon}} + \log n))$ time. Reif and Sun [15] used their approach and introduced a nice variation of Dijkstra's algorithm to obtain an improved $O(\frac{n}{\epsilon} \log \frac{n}{\epsilon} \log \frac{1}{\epsilon})$ bound. Recently, Aleksandrov et [4,5] presented a further improved ϵ-approximation algorithm that runs in $O(\frac{n}{\sqrt{\epsilon}} \log \frac{n}{\epsilon} \log \frac{1}{\epsilon})$ time. (Note that the improvement by a multiplicative factor of $\sqrt{\epsilon}$ is very significant.) The latter algorithm is called bisector ϵ-approximation algorithm in this paper, because it places the Steiner points on the angular bisectors of the triangle faces. The detail of this algorithm will be discussed in the next section. The algorithm [5] improves over the first approach [13] by approximately a factor of n^7, but has a higher dependency on the geometry of the input. The reader is referred to [5] for a full analysis of the algorithm including a detailed discussion of geometric parameters influencing the run-time.

2.1. Bisector Approximation

Our implementation is based on the most recent algorithm, the bisector ϵ-approximation method by Aleksandrov et al. [4,5] sketched next. We discuss the placement of the Steiner points on $\mathcal{P}$ and the edge connectivity used to form G^*. In this scheme the Steiner points are placed on the angular bisectors of each triangle and the edges of E^* are connecting these Steiner points (crossing face boundaries). We define these edges and their costs between any pair of Steiner points (or original vertices) p and q lying on neighbouring bisectors (two bisectors are neighbours if the angles they split share a common edge of $\mathcal{P}$).

For each vertex v, we denote by $E(v)$ the set of edges in the triangles incident to v minus the edges incident to v (the dark outer polygon in Figure 1(1.a)) and by $d(v)$ the minimum Euclidean distance from v to the edges in $E(v)$. The first Steiner point on bisector l of α, is placed at the intersection of l and $\overline{AB}$ where A and B are the intersection of a circle of radius $r' = \epsilon \times r(v) = \epsilon \times \frac{d(v)}{7}$ and the edge of α (Figure 1(1.b)). The remaining Steiner points on l are placed at $|vp_i| = \lambda^i \times |vp_0| \quad for\ i = 1, ..., k$, where $\lambda = (1 + \sqrt{\frac{\epsilon}{2}} \sin \frac{\alpha}{2})$ (Figure 1(1.b)). Similarly we place the Steiner points on all other bisectors.



1 (a-b) Steiner point placement

2 (a-c) Paths used to define edge costs

Figure 1. Defining graph G^*.

There are three cases to consider when connecting a pair of nodes (p, q):

p **and** q **are on different triangles** (Figure 1(2.a)) - A pair of neighbouring bisectors l and l_1 lying on different triangles. The shortest path between p and q bends at a point x on e, so that the angles φ and ψ satisfy the Snell's law: $w \sin\varphi = w_1 \sin\psi$. The cost of the edge ($p$, q) in G^* is equal to $w|px| + w_1|xq|$.

p **and** q **are on different bisectors of the same triangle** (Figure 1(2.b)) - We only add an edge between p and q when $w < w_1$ and there exists two bend points x and y on e, which must satisfy $\varphi = \psi$ and $\sin\varphi = \sin\psi = w/w_1$. The cost of the edge ($p$, q) in G^* is $||pq|| = ||px|| + ||xy|| + ||xq|| = w_1|px| + w|xy| + w_1|yq| = w_1(|px| + |yq|) + w|xy|$.

p **and** q **are on the same bisector** (Figure 1(2.c)) - The situation is similar to the previous one, except that there are two candidates for the shortest path between p and q, one crossing e and the other crossing e_2. We take the shorter resulting distance as weight into G^*.

The discretization step converts the polyhedral surface into a graph G^* upon which one can execute any discrete shortest path algorithm (e.g., Dijkstra's [7]). However, the size of the resulting graph G^* may be quite large which has a significant impact on the running time. In the case of the bisector $\epsilon-$ approximation scheme the number of Steiner points, k, also depends on the geometry of the network. It has been shown that $k = \lfloor \log_\lambda \frac{|l|}{|vp_0|} \rfloor$, where $\lambda = (1 + \sqrt{\frac{\epsilon}{2}} \sin\frac{\alpha}{2})$, and α is the angle that the bisector splits [4] and thus k increases as α gets smaller.

In the next section we present a way to overcome the impact of this increase on the running time of the algorithm. (We also investigated alternate methods to reduce the number of Steiner points at only a small loss in accuracy.)

3. Concurrent Parallelism

Shortest path problems appear to fall into this class of problems are sequential in nature. Using Dijkstra's algorithm, a shortest path $\pi(s, u)$, from s to u, can only be computed after completion of the computation of shortest paths $\pi(s, v)$, where v is any vertex that satisfies $||\pi(s, v)|| < ||\pi(s, u)||$. This manifests itself for parallel shortest paths computations as idle times for a large number of processors. By choosing a good partitioning scheme we can get processors involved more quickly and thus reduce idle times of processors to a certain degree. However, the idle times are still significant, especially when the number of processors increases (the efficiency is roughly $\frac{1}{\sqrt{p}}$ as experimentally determined in [12]).

Existing parallel shortest path implementations deal with concurrent shortest path queries in batch mode, i.e., one query at a time. This implies that queries are waiting even though some processors might be idle. Our proposed concurrent parallelism aims to use these idle cycles to reduce the response time for queries (difference between query's submission time and its output time). The main idea is to overload the processors with queries in such a way that when a query q_i is processed and a processor is idle, it can start to process another query q_j until it is required to resume its work on query q_i. (Note that it requires more resources (e.g., memory).) Allowing the processors to hold and process multiple queries concurrently is called here *concurrent parallelism*. Therefore, our objective is to show that using concurrent parallelism one can process concurrent queries simultaneously to obtain higher speedup and efficiency.

Concurrent parallelism must obey two principles: (1) queries are processed in order of submission, and, (2) each individual query is itself processed as fast as possible in parallel. In our solution to the parallel shortest path problem we adhere to these two principles as follows: (a) each new query is tagged with a query id, and (b) the implementation is based on a parallel shortest path algorithm similar to that of [12].

The parallel shortest path algorithm we implemented, is designed for a distributed memory architecture (e.g., a Beowulf computer). It consists of a preprocessing phase and a computation phase. During the preprocessing phase the graph G^* is generated, partitioned into pages and distributed to the processors using the Multi-dimensional Fixed Partition (MFP) scheme developed in [14]. The computational architecture is based on a manager workers scheme where one processor is designated as a manager and the remaining processors are designated as workers. Although a Beowulf architecture does not have topology, the MFP mapping scheme enables a processor to compute for a given page q the id of the processors which "own" the pages adjacent to q (thus imposing an implicit topology on the Beowulf architecture). During execution time, the manager receives new queries and submits queries to workers. Once a query has been

completed the manager obtains the results and presents the final result to the user.

Each query is processed in parallel by all workers. Our parallel algorithm used to process a single query is build upon a variation of Dijkstra's shortest path algorithm. Each processor executes this algorithm using a priority queue which we denote by SPQueue for each query. This SPQueue uses the travel cost as its key. In our implementation the SPQueue is implemented as a min-heap where the top element of the heap represents the vertex to be processed next in that query.

To meet the first principle of concurrent parallelism, queries must be processed in order of arrival. Thus, if a processor contains data of two different queries, say q_i and q_j, it must process q_i first if q_i was submitted prior to q_j. To process multiple queries simultaneously, each query is assigned a query id in the order of arrival. The query id is used to prioritize the processing order in case a processor contains multiple queries.

In our design, we use one SPQueue for each concurrent query and a priority queue (denoted by QueryHeap) as a container for these SPQueues. At the beginning of each iteration of Dijkstra's algorithm, we only access the SPQueue with highest priority among all SPQueues currently stored in QueryHeap. The priority of a SPQueue in the QueryHeap is determined by a pair (query status, query id). When SPQueue of a query is empty, its status is IDLE, otherwise its status is WORKING. A SPQueue with WORKING status has higher priority in QueryHeap than a SPQueue with IDLE status. Among all SPQueues with the same status, the SPQueue with smaller query id has higher priority. During execution a processor receives messages including (1) update message which contains the new travel cost of vertices, (2) query termination message. An update message is processed using Algorithm 1. When a query termination message is received the processor removes the query from QueryQueue.

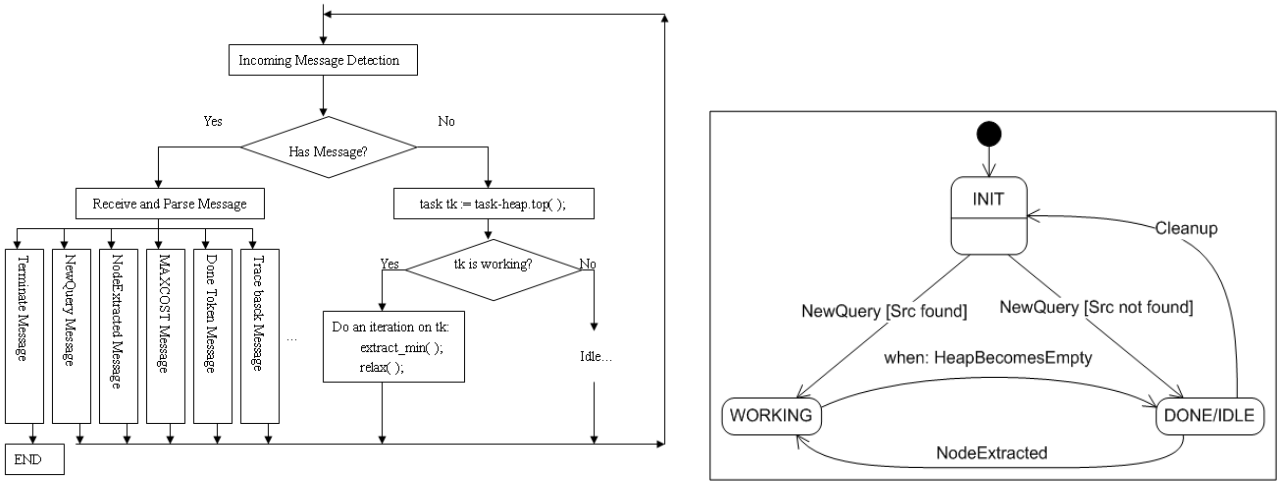
Algorithm 1 ProcessUpdateMessage()

```
1: if Query is not in the QueryQueue then
2:     Create a new SPQueue for the query
3:     Insert updated vertices to the SPQueue
4:     Insert the new query into the QueryQueue
5: else
6:     Locate the query in the QueryQueue
7:     Update the cost of the vertices
8:     Update the SPQueue of the query
9: end if
```

We observe from our experiments (see Section 4) that a relatively small number of concurrent queries are already sufficient to achieve a very good performance for our concurrent shortest path application. A skeleton of the execution of a worker processor is given in Algorithm 2 and depicted in Figure 2. Next we present the testing and the experimental results of concurrent parallelism.

Algorithm 2 ConcurrentSP()

```
1: for as long as there is work to do do
2:     if there is a message then
3:         if update message then
4:             Execute Algorithm 1
5:         else
6:             Remove query from the QueryQueue
7:         end if
8:     end if
9:     Obtain top element of QueryQueue and obtain its SPQueue
10:    Locally run Dijkstra shortest path algorithm of SPQueue
11:    Communicate with other processors to update cost and path information at partition boundaries
12:    if shortest path is found then
13:        Report the length
14:        Remove the query from the QueryQueue
15:    end if
16: end for
```



(1) Control Flow (2) State chart

Figure 2. Worker processor control flow and state chart.

4. Experimental Results

The objective of concurrent parallelism is to take advantage of idle cycles when executing a parallel application. We can only provide an overview of the extensive set of experiments conducted and present the results for random distribution of queries. The reader is referred to the full paper for a detailed discussion of the other (somewhat similar) results.

4.1. Test Setup

We tested the concept of concurrent parallelism using a parallel shortest path application. In our tests we examined a number of factors: parallel machine architecture, number of processors, type of queries, and concurrency level. We conducted our tests on a variety of spatial data and queries.

Machine Architecture - We employed a Symmetric Multi Processing (SMP) architecture - SunFire 6800 Cluster, and a distributed memory architecture - Pentium Xeon-based Beowulf Cluster. The SunFire had 20GB of memory, a 1.3 TB disk storage - Sun StorEdge[tm] T3 Disk array, and twenty 900 MHz UltraSPARC III Cu processors. The communication software was Sun Grid Engine v5.3 Enterprise Edition with Sun MPI v5.0. The Beowulf computer consists of 128 processors: (a) 32 nodes with dual 1.7 GHz Xeon processors, 1 GB RAM per node and 60GB disk (b) 32 nodes with dual 2.0 GHz Xeon processors 1.5 GB RAM per node and 60GB disk. The interconnect network is a Cisco 6509 switch with 1Gb cards for the nodes (the 1.7 GHz nodes use Intel Pro 1000 XT NICs and the 2.0 GHz nodes use on-board GigE interfaces). The OS was Redhat 7.3 using Sun Grid Engine Enterprise Edition with LAM-MPI v6.5.9.

- **The number of processors -** is 1, 4, 9, 16 corresponding to a mesh (torus) of size 1x1, 2x2, 3x3, and 4x4, respectively.
- **Type of queries-** queries are drawn from three query distributions: random distribution, and two cases of clustered distribution (see Figure 3).
  - **Clustered distribution -** tries to mimic spatial queries that relate to particular locations (e.g., downtown). We first randomly chose five vertices to represent five "city centres". Then, for each "city centre", we created a mixture of short, medium and long distance queries using two distance measures: link distance (clustered case 1) and face distance (clustered case 2). Using the link distance measure we conducted a one-to-all shortest path from each "city centre", say vertex s, to all other vertices. Let $d_v(s,u)$ denote the number of vertices along the shortest path from s to u and let w be the vertex such that $d_v(s,w) = \max d_v(s,u), u \in V^*$. A query from vertex s to u is a *short* query if $d_v(s,u) \leq \frac{d_v(s,w)}{3}$. Similarly, a query from vertex s to u is a *medium* (*long*) query if $\frac{d_v(s,w)}{3} < d_v(s,u) < \frac{2d_v(s,w)}{3}$ (if $\frac{2d_v(s,w)}{3} \leq d_v(v,u) \leq d_v(s,w)$). In the case of face distance we replaced $d_v(s,u)$ by $d_f(s,u)$ where $d_f(s,u)$ denotes that number of faces (triangles) intersected by the shortest path from s to u.

    We choose three test sets for each of the distance measures: (a) **Long-paths-dominated test (denoted by 5127) -** each source point is associated with one short query, two median queries and seven long queries;

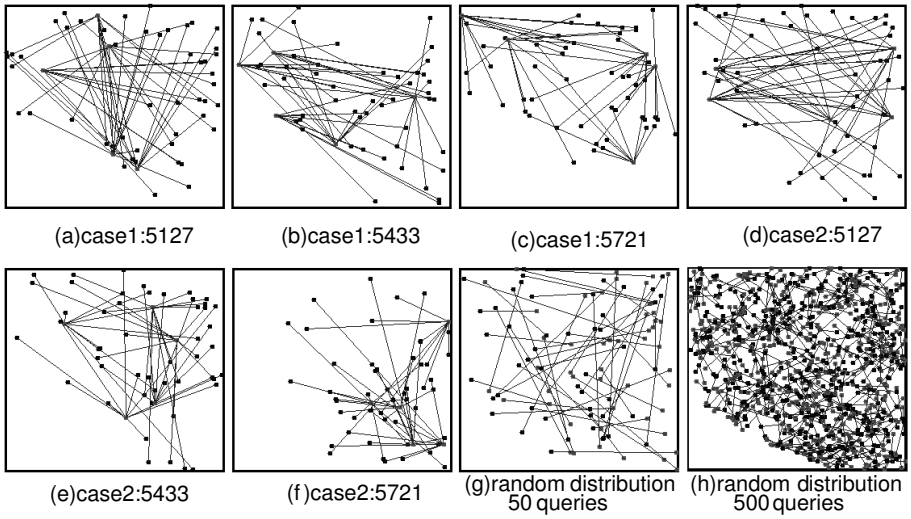


Figure 3. Query distributions - (a), (d) long-paths-dominated test sets, (b), (e) Balanced-distance test sets, (c), (f) short-paths-dominated test sets, and (g), (h) random distribution tests set with 50 queries and 500 short queries, respectively.

(b) **Balanced-distance test set (denoted by 5433) -** each source point is associated with four short queries, three median queries and three long queries; and (c) **Short-paths-dominated set (denoted by 5721) -** each source point is associated with seven short queries, two median queries and one long query. Therefore, each clustered distance measure consists of 50 queries.

– **Random distribution -** Two sets of sets of random queries were used: (a) 50 randomly generated query pairs, and (b) 500 randomly generated short distance queries.

- **Concurrency Level -** different concurrency levels where used 1, 5, 20 and 50.

We used a terrain with 5,000 vertices, 9,799 faces, and 14,798 edges using $\epsilon = 0.1$. The number of edges and vertices grows significantly due to the bisector ϵ-approximation. We chose this terrain to ensure that the data sets and the execution of the algorithm can fit into the memory of a single processor (eliminating any bias due to external memory use). To measure the performance of concurrent parallelism we use the traditional methods of speed up, $S = \frac{T_1}{T_p}$, and efficiency, $E = \frac{S}{p}$, where T_1 is the execution time on a single processor, T_p is the execution time on p processors and p is the number of processors. In addition we introduce a measure called response time to capture how long does it takes to obtain an output to a submitted query. Let T_a and T_f denote the time that a query is submitted and the time that an output is produced, respectively. The response time for the i^{th} query is defined as $T_{r_i} = T_{f_i} - T_{a_i}$, for $i = 1, \ldots, m$, where m is the total number of queries. The average response time $T_{rav} = \frac{1}{m} \sum_{i=1}^{m} T_{r_i}$ is a good indicator of the system performance.

4.2. Results Analysis

Tables 1 and 2 show the results of executing 50 random queries and 500 random short queries, respectively (see figures 3(g) and (h)). We draw attention to: speedup and efficiency, idle time, and response time. Recall that we are interested in the effect of submitting a number of queries simultaneously to the parallel application. Because of space limitations we depict only the results of Table 2 in Figure 4.

I. Speedup and efficiency (Figure 4.c) - here concurrent parallelism significantly improved the performance of the parallel shortest path application. In the case of 500 random short queries, we obtained about 100% improvement over sequential query submission. For example, sequential submission obtained a speed up of close to 3 whereas for 50 concurrent queries, the speed up improved to about 7. When testing 50 random queries we obtained a 50% improvement (for 16 processors the speedup improvement was from 5.8 to 7.5). For 500 short queries we obtain a greater performance improvement because only one or two processors were involved in computing the shortest path due to the small distance between source and destination vertices. Since the 500 short queries were uniformly distributed we obtained better parallel utilization.

II. Idle time and computation time (Figure 4.a) - the idle time of the processors was significantly reduced. This is noticeable in all types of queries and for all processors. E.g., for 500 random queries the idle times of 4 and 16 processors was reduced from 591, and 704 seconds to about 2 and 6.5 seconds, respectively when the number of concurrent queries

Concurrency threshold	processor configuration	COMM	IDLE	COMP	TOTAL	SPEEDUP	efficiency	average response time
50	1x1	0.01	0	3264.64	3264.64	1	100.00%	1592.59
50	2x2	132.16	3.35	1059.39	1194.9	2.73	68.30%	577.47
50	3x3	85.46	25.53	571.84	682.83	4.78	53.12%	333.16
50	4x4	82.59	5.9	347.19	435.69	7.49	46.83%	211.42
20	1x1	0.01	0	3263.2	3263.2	1	100.00%	1594.03
20	2x2	132.45	2.33	1058.32	1193.11	2.74	68.38%	579.50
20	3x3	82.72	22.88	553.43	659.03	4.95	55.02%	332.15
20	4x4	81.94	8.7	343.34	433.98	7.52	47.00%	212.88
5	1x1	0.01	0	3260.74	3260.75	1	100.00%	1587.52
5	2x2	131.29	14.92	1049.57	1195.77	2.73	68.17%	581.27
5	3x3	85.63	41.36	566.75	693.74	4.7	52.23%	337.34
5	4x4	82.25	17.9	340.3	440.45	7.4	46.27%	212.82
1	1x1	0.03	0	3255.88	3255.91	1	100.00%	1591.68
1	2x2	133.42	164.59	1036.51	1334.52	2.44	60.99%	650.08
1	3x3	79.89	318.6	500.1	898.59	3.62	40.26%	431.35
1	4x4	76.26	179.91	305.19	561.36	5.8	36.25%	278.40

Table 1
50 randomly distributed queries (time in seconds)

Concurrency threshold	processor configuration	COMM	IDLE	COMP	TOTAL	SPEEDUP	efficiency	average response time
50	1x1	0.06	0	3090.33	3090.39	1	100%	1517.07
50	2x2	154.32	1.99	1126.33	1282.63	2.41	60.24%	643.62
50	3x3	108.13	8.79	646.46	763.38	4.05	44.98%	379.09
50	4x4	92.31	6.44	352.12	450.87	6.85	42.84%	221.96
20	1x1	0.06	0	3077.31	3077.37	1	100%	1511.51
20	2x2	152.52	1.81	1121.1	1275.42	2.41	60.32%	643.07
20	3x3	107.56	17.77	638.41	763.73	4.03	44.77%	376.07
20	4x4	93.32	11.98	354.47	459.77	6.69	41.83%	228.61
5	1x1	0.06	0	3073.67	3073.73	1	100%	1510.33
5	2x2	152.96	55.41	1105.5	1313.88	2.34	58.49%	666.12
5	3x3	104.05	141.31	592.34	837.69	3.67	40.77%	414.48
5	4x4	90.12	90.89	329.56	510.57	6.02	37.63%	254.44
1	1x1	0.26	0	3085.73	3085.98	1	100%	1516.79
1	2x2	156.24	591.63	1057.34	1805.21	1.71	42.74%	903.78
1	3x3	102.29	935.68	531.35	1569.32	1.97	21.85%	771.71
1	4x4	87.35	704.77	295.33	1087.45	2.84	17.74%	544.72

Table 2
500 randomly distributed short queries (time in seconds)

grew from 1 to 50. Note, that this reduction in idle times increased processor utilization even though some of the work may not have contributed to an increased performance.

III. Average response time (Figure 4.b) - the response time measure shows a significant reduction in the time a user must wait for output. E.g., in the case of 500 random short queries the response time was cut by more than 50% from 544 seconds to 221 seconds for 16 processors.

We find it important to observe that the level of concurrency does not have to be large in order to obtain a significant improvement in performance. Note that, although there is a continued improvement as the number of concurrent queries grows from 5 to 50, it is not as dramatic as from 1 to 5 concurrent queries. Although, this may be somewhat counter intuitive it can be explained when examining the idle time of the processors (e.g., Figure 4(1.a)). The idle time drops 7-10 fold as the number of concurrent queries grows from 1 to 5.

5. Conclusion

In this paper we present a new paradigm of parallelism - concurrent parallelism where both tasks and executions of individual tasks are parallelized. Through experimental results we showed that concurrent parallelism significantly reduces processor idle times and improves over-all performance by about 15%-50%. By using concurrency we were able to obtain efficiencies of up to 86%, 76% and 72% for 4, 9, 16 processors, respectively on the Sunfire, and 88%, 79% and 65% for 4, 9, and 16 processors, respectively on the Beowulf. These efficiencies are much higher than those obtained without concurrency (61.0%, 40%, and 36% for 4, 9, 16 processors, respectively on the Beowulf).

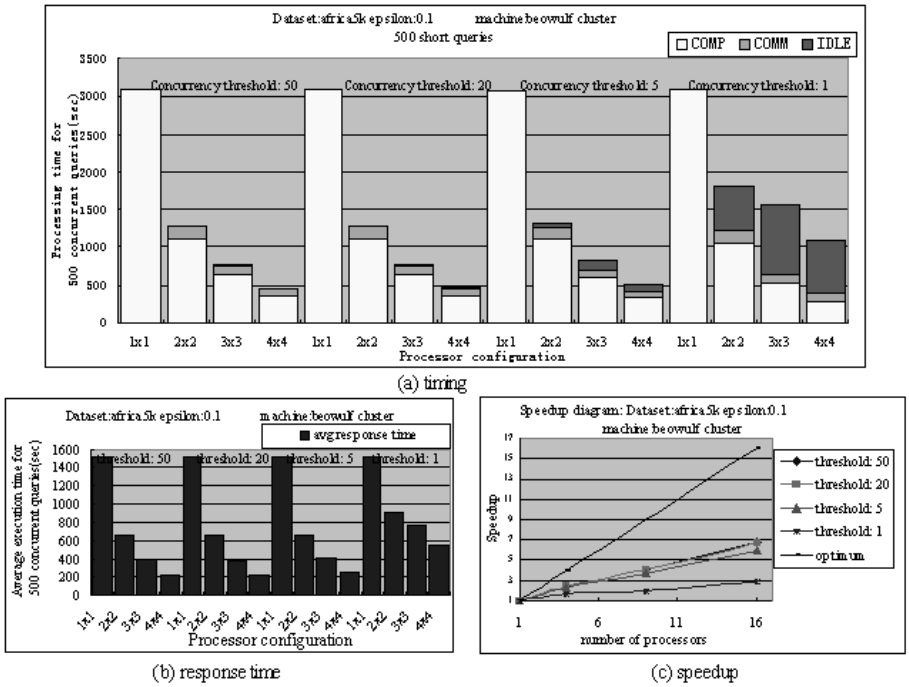


Figure 4. Results of 500 random short distributed queries with different concurrency thresholds.

References

[1] P. Adamson and E. Tick. Greedy partitioned algorithms for the shortest-path problem. *International Journal of Parallel Programming*, 20(4):271–298, 1991.

[2] L. Aleksandrov, M. Lanthier, A. Maheshwari, and J.-R. Sack. An epsilon-approximation algorithm for weighted shortest paths on polyhedral surfaces. In *SWAT: Scandinavian Workshop on Algorithm Theory*, 1998.

[3] L. Aleksandrov, A. Maheshwari, and J.-R. Sack. Approximation algorithms for geometric shortest path problems. In *the 32nd Annual ACM Symposium on Theory of Computing*, pages 286–295, 2000.

[4] L. Aleksandrov, A. Maheshwari, and J.-R. Sack. An improved approximation algorithms for computing geometric shortest paths. In *Foundations of Computation Theory*, pages 246–257, 2003.

[5] L. Aleksandrov, A. Maheshwari, and J.R. Sack. Determining approximate shortest paths on weighted polyhedral surfaces. *Journal of the ACM (JACM)*, 52:25–53, January 2005.

[6] D.P. Bertsekas, F. Guerriero, and R. Musmanno. Parallel asynchronous label-correcting methods for shortest paths. *Jounral of Optimization Theory and Applications*, 88(2):297–320, 1996.

[7] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1(1):269–271, 1959.

[8] M. Hribar, V. Taylor, and D. Boyce. Choosing a shortest path algorithm. Technical Report CSE-95-004, Northwestern University, 1995.

[9] M. Hribar, V. Taylor, and D. Boyce. Performance study of parallel shortest path algorithms: Characteristics of good decompositions. In *13th Annual conference on Intel Supercomputers User Group*, Albuquerque, NM, 1997.

[10] M. Hribar, V. Taylor, and D. Boyce. Reducing the idle time of parallel shortest path algorithms. Technical Report CPDC-TR-9803-016, Northwestern University, 1998.

[11] M. Lanthier, A. Maheshwari, and J.-R. Sack. Approximating weighted shortest paths on polyhedral surfaces. *Algorithmica*, 30(4):527–562, 2001.

[12] M. Lanthier, D. Nussbaum, and J.-R. Sack. Parallel implementation of geometric shortest path algorithms. *Parallel Computing*, 29:1445–1479, 2003.

[13] J.S.B. Mitchell and C.H. Papadimitriou. The weighted region problem: Finding shortest paths through a weighted planar subdivision. *Journal of ACM*, 38:18–73, January 1991.

[14] D. Nussbaum. *Parallel spatial modelling*. PhD thesis, Carleton University, Ottawa, Canada, 2001.

[15] J.H Reif and Z. Sun. An efficient approximation algorithm for weighted region shortest path problem. In *4th workshop on Algorithmic Foundations of Robotics(WAFR2000)*, pages 191–203, 2000.

[16] J.L. Träff. An experimental comparison of two distributed single-source shortest path algorithms. *Parallel Computing*, 21:1505–1532, 1995.

[17] M. Ziegelmann. *Constrained Shortest Paths and Related Problems*. PhD thesis, Max-Planck Institut für Informatik (submitted at Universität des Saarlandes), Saarbrücken, Germany, 2001.